Service Design Overview

This design outlines a Java-based microservice using Spring Boot for simplicity, as it's a popular framework for building scalable services. The service will:

- **Fetch files from Amazon S3**: Daily scheduled task to read "plans" and "rates" files (assuming CSV/JSON formats for simplicity).
- **Save to Mini Storage**: I'll interpret "mini storage" as MinIO (an S3-compatible object storage). Files are copied from S3 to MinIO for local/isolated processing.
- **Process files nightly**: A separate scheduled task to process the saved files (e.g., parse, validate, transform, and store results in a database or output).
- **External API Calls**: Configurable option during processing (e.g., enrich data by calling external APIs like rate APIs).
- **Configurable Adapters**: Use Adapter Pattern for pluggable storage (S3/MinIO) and data source adapters.
- **Processor Pattern**: Use Strategy Pattern for configurable processors (e.g., different processing logic for plans vs. rates).
- **Expose APIs**: REST endpoints to trigger manual fetches/processes, query status, or configure adapters/processors.

Assumptions:

- Files are in S3 buckets with prefixes like `plans/` and `rates/`.
- Processing involves basic operations (e.g., parsing CSV, calculating aggregates); can be extended.
- Use AWS SDK for S3, MinIO SDK for storage.
- Scheduling via Spring's `@Scheduled`.
- Database: Optional H2/PostgreSQL for processed results.
- Configurability: Via application.yml and Spring beans.
- Error Handling: Basic retries and logging with SLF4J.
- Security: Assume API secured with Spring Security (e.g., JWT), but not detailed here.

High-Level Architecture

- **Components**:
    1. **Fetcher**: Scheduled to pull files from S3 and save to MinIO.
    2. **Processor**: Scheduled to process files from MinIO, with optional external API calls.
    3. **Adapters**: Interfaces for storage (e.g., S3Adapter, MinIOAdapter) and data sources.
    4. **Processors**: Strategy interface for different processing logic (e.g., PlansProcessor, RatesProcessor).
    5. **API Layer**: REST controllers for manual triggers and configs.
- **Patterns**:
    1. **Adapter Pattern**: Abstracts storage operations (read/write) so S3/MinIO can be swapped/configured.
    2. **Strategy Pattern**: For processors, allowing runtime configuration (e.g., via API or config file).
- **Flow**:
    1. Daily: Fetch files from S3 → Adapt and save to MinIO.
    2. Nightly: Read from MinIO → Process using configured strategy → Optional external API call → Store results.
    3. APIs: Expose endpoints for on-demand operations.

Text-based diagram:

```
[External API] <-- Optional Call
            |
[Scheduled Fetcher] --> [S3 Adapter] --> S3 Bucket (plans/rates files)
            |
            v
[MinIO Adapter] <-- Save Files --> MinIO Storage
            |
[Scheduled Processor] --> Read Files --> [Processor Strategy] --> Process & Store Results (DB)
            |
[REST APIs] <-- Trigger/Manage --> Fetcher/Processor/Config
```

Dependencies (pom.xml Snippet)

Use Maven for build. Key dependencies:

```
        org.springframework.boot
        spring-boot-starter-web


        org.springframework.boot
        spring-boot-starter-scheduler


        software.amazon.awssdk
        s3
        2.20.0


        io.minio
        minio
        8.5.0


        org.springframework.boot
        spring-boot-starter-data-jpa


        org.apache.httpcomponents
        httpclient
```

Configuration (application.yml)

Make adapters and processors configurable via properties.

```
app:
  storage:
    source: s3  # or other adapters
    target: minio
  processor:
    type: default  # e.g., plans, rates, custom
  s3:
```

```
    bucket: my-s3-bucket
    access-key: ${AWS_ACCESS_KEY}
    secret-key: ${AWS_SECRET_KEY}
    region: us-east-1
  minio:
    url: http://localhost:9000
    access-key: minioadmin
    secret-key: minioadmin
    bucket: my-minio-bucket
  scheduler:
    fetch-cron: 0 0 9 * * ?  # Daily at 9 AM
    process-cron: 0 0 0 * * ?  # Nightly at midnight
  external-api:
    enabled: false
    url: https://api.example.com/rates
    api-key: ${API_KEY}
```

Core Interfaces and Patterns

Adapter Pattern for Storage

Interface for storage operations, with implementations for S3 and MinIO.

```
public interface StorageAdapter {
    List listFiles(String prefix); // e.g., "plans/" or "rates/"
    InputStream readFile(String fileKey);
    void writeFile(String fileKey, InputStream content);
}

// S3 Implementation
@Component("s3Adapter")
public class S3Adapter implements StorageAdapter {
    private final S3Client s3Client;
    private final String bucket;

    // Constructor injection via @Autowired and config properties

    @Override
    public List listFiles(String prefix) {
        // Use s3Client.listObjectsV2() to list keys
        return ...;
    }

    @Override
    public InputStream readFile(String fileKey) {
        return
s3Client.getObject(GetObjectRequest.builder().bucket(bucket).key(fileKey).build());
```

```
    }

    @Override
    public void writeFile(String fileKey, InputStream content) {
        // Not typically used for S3 as source, but implement if needed
    }
}

// MinIO Implementation (similar to S3, using MinioClient)
@Component("minioAdapter")
public class MinIOAdapter implements StorageAdapter {
    private final MinioClient minioClient;
    private final String bucket;

    // Implement listFiles, readFile, writeFile using MinioClient
}
```

Strategy Pattern for Processors

Interface for processing logic, configurable at runtime.

```
public interface FileProcessor {
    void process(InputStream fileContent, String fileType); // fileType: "plans" or "rates"
    // Optional: Return processed data or throw exceptions
}

// Default Processor (e.g., parse CSV and save to DB)
@Component("defaultProcessor")
public class DefaultFileProcessor implements FileProcessor {
    @Override
    public void process(InputStream fileContent, String fileType) {
        // Use CSVParser or Jackson to parse
        // Example: Calculate rates, save to repository
        // If external API enabled, call it here (see below)
    }
}

// Plans-Specific Processor
@Component("plansProcessor")
public class PlansProcessor implements FileProcessor {
    // Custom logic for plans files
}

// Rates-Specific Processor (similar)
```

External API Calls

Add a utility for optional calls, injected into processors.

```
@Service
public class ExternalApiService {
    private final RestTemplate restTemplate; // or WebClient for reactive
    private final boolean enabled;
    private final String url;
    private final String apiKey;

    // Constructor from config

    public String callApi(String query) {
        if (!enabled) return null;
        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer " + apiKey);
        // Use restTemplate.getForObject(url + "?q=" + query, String.class);
        return response;
    }
}
```

In processors: `externalApiService.callApi("some-rate-query");` for enrichment.

Scheduled Tasks

Use Spring's scheduling.

```
@Service
public class FileFetcherService {
    private final StorageAdapter sourceAdapter; // S3
    private final StorageAdapter targetAdapter; // MinIO

    @Autowired
    public FileFetcherService(@Qualifier("${app.storage.source}Adapter") StorageAdapter
source,
                              @Qualifier("${app.storage.target}Adapter") StorageAdapter target) {
        this.sourceAdapter = source;
        this.targetAdapter = target;
    }

    @Scheduled(cron = "${app.scheduler.fetch-cron}")
    public void fetchDaily() {
        List files = sourceAdapter.listFiles("plans/");
        files.addAll(sourceAdapter.listFiles("rates/"));
        for (String file : files) {
            InputStream content = sourceAdapter.readFile(file);
            targetAdapter.writeFile(file, content); // Copy to MinIO
```

```
        }
        // Log success/errors
    }
}

@Service
public class FileProcessorService {
    private final StorageAdapter storageAdapter; // MinIO
    private final FileProcessor processor; // Configured strategy

    @Autowired
    public FileProcessorService(@Qualifier("${app.storage.target}Adapter") StorageAdapter
storage,
                                @Qualifier("${app.processor.type}Processor") FileProcessor proc) {
        this.storageAdapter = storage;
        this.processor = proc;
    }

    @Scheduled(cron = "${app.scheduler.process-cron}")
    public void processNightly() {
        List files = storageAdapter.listFiles(""); // All files in MinIO
        for (String file : files) {
            InputStream content = storageAdapter.readFile(file);
            String type = file.contains("plans") ? "plans" : "rates";
            processor.process(content, type);
            // Optional: Delete after processing
        }
    }
}
```

Exposed APIs (REST Controller)

```
@RestController
@RequestMapping("/api")
public class ServiceController {
    private final FileFetcherService fetcher;
    private final FileProcessorService processor;

    // Autowired constructor

    @PostMapping("/fetch")
    public ResponseEntity manualFetch() {
        fetcher.fetchDaily();
        return ResponseEntity.ok("Fetch triggered");
    }

    @PostMapping("/process")
```

```
    public ResponseEntity manualProcess() {
        processor.processNightly();
        return ResponseEntity.ok("Process triggered");
    }

    @GetMapping("/status")
    public ResponseEntity getStatus() {
        // Return fetch/process logs or status
        return ResponseEntity.ok("Status: OK");
    }

    @PutMapping("/config/processor")
    public ResponseEntity updateProcessor(@RequestParam String type) {
        // Dynamically switch processor bean (use ApplicationContext or reload config)
        // For simplicity, restart service or use @RefreshScope
        return ResponseEntity.ok("Processor updated to " + type);
    }

    // Similar for adapters
}
```

Extensibility and Best Practices

- **Configurability**: Use `@Qualifier` with property values to switch adapters/processors without code changes.
- **Error Handling**: Wrap operations in try-catch, use retries with Resilience4j.
- **Testing**: Unit test adapters/processors; integration test with LocalStack (for S3/MinIO mocking).
- **Scaling**: For production, use Kubernetes for scheduling, add monitoring with Prometheus.
- **Security**: Secure S3/MinIO creds with AWS Secrets Manager or Spring Cloud Config.
- **Improvements**: Add file versioning, multi-threaded processing, or Kafka for event-driven flow if needed.

This design is modular, allowing easy swapping of components. If “mini storage” means something else (e.g., local FS), the adapter can be extended accordingly.

Enhancing an Existing Plans Service

Assuming your existing “plans service” is a Java-based Spring Boot application that already handles some core logic (e.g., managing or processing plan data in a database or via manual triggers), you can enhance it by integrating the features from the proposed design. This includes automated S3 file fetching, storage in MinIO (or similar “mini storage”), nightly processing with configurable adapters and processors, optional external API calls, and exposed REST APIs. The key is to add these as modular extensions without disrupting the current codebase, leveraging Spring’s dependency injection and configuration for seamless integration.

The Adapter Pattern will make storage (S3/MinIO) pluggable, and the Strategy Pattern will allow flexible processing logic. This ensures the enhancements are configurable and extensible. Below is a step-by-step guide to incorporate these into your existing service.

Step 1: Add Required Dependencies

Update your `pom.xml` to include libraries for S3, MinIO, scheduling, and external API calls. If your service already uses Spring Boot, these will integrate easily.

```
software.amazon.awssdk
s3
2.20.0


io.minio
minio
8.5.0


org.springframework.boot
spring-boot-starter-scheduler


org.springframework.boot
spring-boot-starter-web


org.springframework.boot
spring-boot-starter-data-jpa
```

Run `mvn clean install` to refresh dependencies.

Step 2: Configure Properties

Extend your `application.yml` (or `application.properties`) with sections for storage, processors, scheduling, and external APIs. This makes everything configurable without code changes.

```
app:
  storage:
    source: s3  # Source adapter (e.g., S3 for fetching)
    target: minio  # Target adapter (e.g., MinIO for storage)
  processor:
    type: default  # Or 'plans' for custom logic
  s3:
    bucket: my-s3-bucket
    access-key: ${AWS_ACCESS_KEY}  # Use env vars for security
    secret-key: ${AWS_SECRET_KEY}
    region: us-east-1
  minio:
    url: http://localhost:9000  # Or your MinIO instance
    access-key: minioadmin
    secret-key: minioadmin
    bucket: my-minio-bucket
  scheduler:
    fetch-cron: 0 0 9 * * ?  # Daily fetch at 9 AM
    process-cron: 0 0 0 * * ?  # Nightly process at midnight
  external-api:
    enabled: false
    url: https://api.example.com/rates  # Your external API endpoint
    api-key: ${API_KEY}
```

If your existing service has overlapping configs (e.g., DB settings), merge them.

Step 3: Implement Configurable Adapters (Adapter Pattern)

Add the `StorageAdapter` interface and implementations. This abstracts file operations, allowing you to swap S3 for fetching and MinIO for storage.

- Create the interface:

```
public interface StorageAdapter {
    List listFiles(String prefix);  // e.g., "plans/" or "rates/"
```

```
    InputStream readFile(String fileKey);
    void writeFile(String fileKey, InputStream content);
}
```

- S3 Adapter (for source):

```
@Component("s3Adapter")
public class S3Adapter implements StorageAdapter {
    private final S3Client s3Client;
    private final String bucket;

    @Autowired
    public S3Adapter(@Value("${app.s3.access-key}") String accessKey,
                     @Value("${app.s3.secret-key}") String secretKey,
                     @Value("${app.s3.region}") String region,
                     @Value("${app.s3.bucket}") String bucket) {
        this.s3Client = S3Client.builder()

.credentialsProvider(StaticCredentialsProvider.create(AwsBasicCredentials.create(accessKe
y, secretKey)))
                .region(Region.of(region))
                .build();
        this.bucket = bucket;
    }

    @Override
    public List listFiles(String prefix) {
        ListObjectsV2Request request =
ListObjectsV2Request.builder().bucket(bucket).prefix(prefix).build();
        return
s3Client.listObjectsV2(request).contents().stream().map(S3Object::key).collect(Collectors.to
List());
    }

    @Override
    public InputStream readFile(String fileKey) {
        return
s3Client.getObject(GetObjectRequest.builder().bucket(bucket).key(fileKey).build());
    }

    @Override
    public void writeFile(String fileKey, InputStream content) {
        // Implement if needed for future use
    }
}
```

- MinIO Adapter (for target, similar structure):

```
@Component("minioAdapter")
public class MinIOAdapter implements StorageAdapter {
    private final MinioClient minioClient;
    private final String bucket;

    @Autowired
    public MinIOAdapter(@Value("${app.minio.url}") String url,
                        @Value("${app.minio.access-key}") String accessKey,
                        @Value("${app.minio.secret-key}") String secretKey,
                        @Value("${app.minio.bucket}") String bucket) {
        this.minioClient = MinioClient.builder()
                .endpoint(url)
                .credentials(accessKey, secretKey)
                .build();
        this.bucket = bucket;
    }

    // Implement listFiles, readFile, writeFile using minioClient (e.g., minioClient.listObjects, getObject, putObject)
}
```

Inject these into your services using `@Qualifier("${app.storage.source}Adapter")` for dynamic selection.

Step 4: Implement Processors (Strategy Pattern) and External API Calls

Add the `FileProcessor` interface for pluggable processing. Integrate with your existing plans logic (e.g., call existing methods during processing).

- Interface:

```
public interface FileProcessor {
    void process(InputStream fileContent, String fileType);  // Enhance with your existing plans processing
}
```

- Default Implementation (integrate external calls):

```
@Component("defaultProcessor")
public class DefaultFileProcessor implements FileProcessor {
    private final ExternalApiService externalApiService;  // See below
```

```
    @Autowired
    public DefaultFileProcessor(ExternalApiService externalApiService) {
        this.externalApiService = externalApiService;
    }

    @Override
    public void process(InputStream fileContent, String fileType) {
        // Parse file (e.g., CSV with OpenCSV or Jackson)
        // Call your existing plans service methods here, e.g.,
existingPlansService.updatePlans(parsedData);
        if (externalApiService.isEnabled()) {
            String apiResponse = externalApiService.callApi("query-based-on-file");  // Enrich
data
            // Merge apiResponse into processing
        }
        // Save results to DB or your existing storage
    }
}
```

- External API Service:

```
@Service
public class ExternalApiService {
    private final RestTemplate restTemplate = new RestTemplate();
    @Value("${app.external-api.enabled}")
    private boolean enabled;
    @Value("${app.external-api.url}")
    private String url;
    @Value("${app.external-api.api-key}")
    private String apiKey;

    public boolean isEnabled() { return enabled; }

    public String callApi(String query) {
        if (!enabled) return null;
        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer " + apiKey);
        HttpEntity entity = new HttpEntity<>(headers);
        return restTemplate.exchange(url + "?q=" + query, HttpMethod.GET, entity,
String.class).getBody();
    }
}
```

Add custom processors (e.g., `@Component("plansProcessor")`) if needed, extending your current logic.

Step 5: Add Scheduled Tasks

Create services for fetching and processing, integrating with your existing components.

- Fetcher Service:

```
@Service
@EnableScheduling
public class FileFetcherService {
    private final StorageAdapter sourceAdapter;
    private final StorageAdapter targetAdapter;

    @Autowired
    public FileFetcherService(@Qualifier("${app.storage.source}Adapter") StorageAdapter
source,
                              @Qualifier("${app.storage.target}Adapter") StorageAdapter target) {
        this.sourceAdapter = source;
        this.targetAdapter = target;
    }

    @Scheduled(cron = "${app.scheduler.fetch-cron}")
    public void fetchDaily() {
        // Fetch plans/rates from S3, save to MinIO
        List files = sourceAdapter.listFiles("plans/");
        files.addAll(sourceAdapter.listFiles("rates/"));
        for (String file : files) {
            InputStream content = sourceAdapter.readFile(file);
            targetAdapter.writeFile(file, content);
        }
    }
}
```

- Processor Service (similar to above, using `@Scheduled` and injecting `FileProcessor` via `@Qualifier("${app.processor.type}Processor")`).

Enable scheduling in your main application class with `@EnableScheduling`.

Step 6: Expose APIs

If your service doesn't have REST endpoints, add a controller. Otherwise, extend the existing one.

```
@RestController
@RequestMapping("/api/plans")
public class PlansController {
    private final FileFetcherService fetcher;
```

```
    private final FileProcessorService processor;

    @Autowired
    public PlansController(FileFetcherService fetcher, FileProcessorService processor) {
        this.fetcher = fetcher;
        this.processor = processor;
    }

    @PostMapping("/fetch")
    public String manualFetch() {
        fetcher.fetchDaily();
        return "Fetch completed";
    }

    @PostMapping("/process")
    public String manualProcess() {
        processor.processNightly();
        return "Processing completed";
    }

    // Add more: e.g., /config to update processor type dynamically
}
```

Step 7: Integration and Testing

- **Integrate with Existing Logic**: In processors, invoke your current plans methods (e.g., inject your existing `PlansService` and call `updatePlansFromFile()`).
- **Error Handling**: Add try-catch, logging (SLF4J), and retries (e.g., with Resilience4j).
- **Security**: Secure creds with Spring Vault or env vars; add authentication to APIs if needed.
- **Testing**: Write unit tests for adapters/processors (mock with Mockito). Use integration tests with Testcontainers for MinIO/S3 mocks.
- **Deployment**: Restart the service after changes. For production, monitor with Actuator endpoints.

Benefits

- **Modularity**: Adapters/processors allow switching storage or logic via config (no redeploy).
- **Automation**: Scheduling reduces manual work.

- **Extensibility**: Easily add more file types or API integrations.
- **Minimal Disruption**: These additions bolt onto your existing service without refactoring core code.

If your existing service has specific constraints (e.g., non-Spring, different DB), provide more details for tailored advice.

Security Aspects of the Service

The service design incorporates several security best practices to protect data, credentials, and operations. Below is a comprehensive list of key security aspects, focusing on authentication, authorization, data handling, and compliance. These are built on Spring Boot's security features and cloud-native principles:

- **Credential Management**:
    - S3 and MinIO access keys/secrets are never hardcoded. Instead, they are injected via environment variables or external secrets managers (e.g., AWS Secrets Manager, HashiCorp Vault, or Spring Cloud Config). This prevents exposure in code repositories.
    - Use least-privilege IAM roles for S3 access (e.g., read-only permissions for fetching files). For MinIO, apply similar role-based access control (RBAC) policies.
    - Rotate credentials periodically using automated tools like AWS Secrets Manager's rotation feature.
- **Data Encryption**:
    - Enable server-side encryption (SSE) on S3 buckets using AWS-managed keys (SSE-S3) or customer-managed keys (SSE-KMS) to encrypt files at rest.
    - For MinIO, configure SSE with MinIO's built-in support or integrate with KMS for encryption at rest.
    - Use HTTPS/TLS for all network communications: Enforce SSL for S3/MinIO endpoints and external API calls (e.g., via RestTemplate with SSL configuration).
    - In-transit encryption for file transfers between S3 and MinIO.
- **Access Controls**:

    - API endpoints are secured with Spring Security (e.g., JWT or OAuth2). For example, require authentication for manual fetch/process triggers (/api/fetch, /api/process).
    - Role-based authorization: Admins can update configs (e.g., processor type), while read-only users can only check status.
    - Bucket policies: Restrict S3/MinIO buckets to specific IP ranges or VPC endpoints to prevent public access.
    - Input validation: Sanitize file keys and prefixes to avoid path traversal attacks.
- **Auditing and Logging**:
    - Use SLF4J with Logback or ELK stack for logging all operations (e.g., file fetches, processes, errors). Mask sensitive data like credentials in logs.
    - Enable AWS CloudTrail for S3 access audits and MinIO's audit logs for object operations.
    - Monitor for anomalies using tools like Spring Boot Actuator with Micrometer and Prometheus.
- **Error Handling and Resilience**:
    - Implement circuit breakers (e.g., Resilience4j) for external calls (S3, MinIO, APIs) to prevent cascading failures.
    - Secure exception handling: Avoid leaking stack traces or sensitive info in API responses.
    - Rate limiting on APIs to prevent DDoS-like abuse.
- **Compliance and Best Practices**:
    - Follow OWASP guidelines for web services (e.g., secure headers like CSP, X-XSS-Protection).
    - Container security: If deployed on Docker/Kubernetes, scan images for vulnerabilities and use secrets volumes for creds.
    - Regular security scans: Integrate tools like SonarQube or Dependabot for code vulnerabilities.
    - Data privacy: If plans/rates files contain PII, ensure GDPR/HIPAA compliance by anonymizing data during processing.

These aspects can be audited and enhanced based on your organization's security policies.

How S3 Credentials Are Stored Per Environment

To handle different environments (e.g., dev, test, staging, prod), use Spring Profiles and externalized configuration for secure, environment-specific credential storage. This avoids mixing dev/prod secrets and allows easy switching via `--spring.profiles.active=prod`.

- **Spring Profiles Integration**:
    - Create profile-specific YAML files: `application-dev.yml`, `application-prod.yml`, etc.

In each, define S3 creds placeholders like:

```
 # application-prod.yml
app:
  s3:
    access-key: ${AWS_ACCESS_KEY_PROD}
    secret-key: ${AWS_SECRET_KEY_PROD}
    region: us-east-1
```

    -
    - Resolve placeholders from environment variables (e.g., set `AWS_ACCESS_KEY_PROD` in your CI/CD pipeline or server env).

- **Secrets Management Tools**:
    - **AWS Secrets Manager (Recommended for AWS Environments)**: Store creds as secrets (e.g., `/prod/s3/access-key`). In Spring, use `@Value("${app.s3.access-key}")` and fetch via AWS SDK integration (add `spring-cloud-starter-aws-secrets-manager-config` dependency). Secrets are versioned, rotated, and accessed with IAM roles—no direct storage in code or files.
    - **HashiCorp Vault**: Integrate via Spring Cloud Vault. Vault pulls secrets dynamically at runtime, supporting per-environment namespaces (e.g., `dev/s3`, `prod/s3`).
    - **Kubernetes Secrets**: If deployed on K8s, store creds as K8s Secrets and mount as env vars or volumes. Use Sealed Secrets for encryption.
    - **Environment Variables**: Fallback for local dev—set via `.env` files (ignored in Git) or IDE run configs. For prod, use cloud provider's env injection (e.g.,

AWS ECS task definitions).

- **Best Practices**:
    - Never commit secrets to Git—use `.gitignore` for local files.
    - Use IAM roles instead of access keys where possible (e.g., EC2 instance roles for S3 access).
    - Audit access: Log who/when creds are retrieved.
    - Testing: In dev, use temporary creds or LocalStack to mock S3 without real secrets.

This setup ensures creds are isolated, reducing breach risks across environments.

File Purge Policy from MinIO

The “mini” refers to MinIO storage. A purge policy automates file cleanup to manage storage costs, comply with data retention rules, and maintain performance. Implement it as part of the processing flow or via MinIO’s lifecycle management.

- **Default Policy Recommendation**:
    - **Immediate Purge After Processing**: In the `FileProcessorService`, delete files from MinIO immediately after successful processing (e.g., via `minioClient.removeObject()` in `MinIOAdapter`). This is suitable for transient data like daily plans/rates files.
        - Condition: Only purge if processing succeeds (no errors) to allow retries.
        - Retention: 0 days post-processing—ideal for non-auditable data.
    - **Time-Based Retention**: Keep files for a configurable period (e.g., 7 days) for debugging or audits. Use MinIO’s Object Lifecycle Management (via `minioClient.setBucketLifecycle()`) to auto-delete objects older than X days.
        - Example Config: In `application.yml`, add `app.minio.purge-days: 7`. Schedule a daily cron to apply lifecycle rules or scan and delete.
    - **Size/Quota-Based**: Monitor bucket size; purge oldest files if exceeding a threshold (e.g., via custom script in a scheduled task).

- **Implementation in Code**:
    - Extend `MinIOAdapter` with a `deleteFile(String fileKey)` method.
    - In `processNightly()`: After `processor.process(...)`, call `storageAdapter.deleteFile(file)`.
    - Error Handling: If delete fails, log and retry (e.g., with exponential backoff).
    - Auditing: Log purges with timestamps and reasons.
- **Configuration**:
    - Make it toggleable: `app.minio.purge-enabled: true` and `app.minio.purge-policy: immediate` (or `time-based`).
    - Compliance: For regulated data, retain for longer (e.g., 30 days) and use versioning to allow restores.

This policy can be adjusted based on your storage needs—start with immediate purge for efficiency, and add retention if required for compliance.

High-Level Components and Flow
• External Sources: AWS S3 (source of plans/rates files), External APIs (for data enrichment).
• Core Service: Spring Boot app with adapters (S3Adapter, MinIOAdapter), processors (e.g., DefaultFileProcessor), schedulers (FileFetcherService, FileProcessorService).
• Storage: MinIO for temporary file storage.
• Outputs: Processed data to DB, REST APIs for triggers and configs.
• Security: Credential management via secrets, encryption, access controls.

@startuml title Sequence Diagram: Daily File Fetch Flow
participant Scheduler as “Spring Scheduler” participant Service as “Plans Service (DEP)” participant S3 as “AWS S3” participant MinIO as “MinIO Storage”
Scheduler -> Service: Trigger fetchDaily() at cron time activate Service Service -> S3: listFiles(“plans/”) via S3Adapter S3 –> Service: List of file keys Service -> S3: readFile(fileKey) for each file S3 –> Service: InputStream content Service -> MinIO: writeFile(fileKey, content) via MinIOAdapter MinIO –> Service: Success deactivate Service
@enduml
@startuml title Sequence Diagram: Nightly File Process Flow
participant Scheduler as “Spring Scheduler” participant Service as “Plans Service (DEP)” participant MinIO as “MinIO Storage” participant ExtAPI as “External API” participant DB as “Database/Output”
Scheduler -> Service: Trigger processNightly() at cron time activate Service Service -> MinIO: listFiles(””) via MinIOAdapter MinIO –> Service: List of file keys Service -> MinIO: readFile(fileKey) for each file MinIO –> Service: InputStream content Service -> Service:

Determine fileType (plans/rates) Service -> Service: process(content, fileType) via FileProcessor (Strategy) opt If external-api.enabled Service -> ExtAPI: callApi(query) via ExternalApiService ExtAPI –> Service: Response (enrich data) end Service -> DB: Save processed results (integrate with existing plans logic) DB –> Service: Success Service -> MinIO: deleteFile(fileKey) if purge policy = immediate MinIO –> Service: Success deactivate Service
@enduml
@startuml title Sequence Diagram: Manual API Trigger Flow (e.g., Fetch)
participant User as “User/Client” participant Ingress as “Ingress (NGINX)” participant ServiceSvc as “Service (ClusterIP)” participant Pod as “Plans Pod (DEP)”
User -> Ingress: POST /api/fetch (HTTP Request) Ingress -> ServiceSvc: Route to service ServiceSvc -> Pod: Load balance to pod activate Pod Pod -> Pod: Execute fetchDaily() manually note right: Similar to fetch flow: Interact with S3 and MinIO Pod –> ServiceSvc: Response “Fetch completed” ServiceSvc –> Ingress: Forward response Ingress –> User: HTTP Response deactivate Pod
@enduml
@startuml title Sequence Diagram: File Purge via CronJob (Time-Based)
participant K8sCron as “K8s CronJob (minio-purge)” participant MinIO as “MinIO Storage” participant MC as “MinIO Client (mc)”
K8sCron -> MC: Run at scheduled time (e.g., 2 AM) activate MC MC -> MinIO: Set alias and auth with secrets MinIO –> MC: Connected MC -> MinIO: ilm add –expiry-days 7 (lifecycle policy) MinIO –> MC: Policy applied MC -> MinIO: rm –recursive –older-than 7d /my-bucket/ MinIO –> MC: Files purged deactivate MC
@enduml